Avalanche: A Framework for Parallel and Distributed Computation

by

Uriel Schafer

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

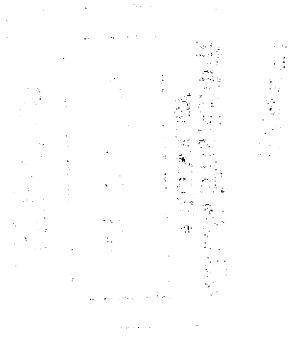
Feb 2002

Author .. ..
Department of Electrical Engineering and Computer Science
Feb 1, 2002

Certified by.. .....
Kath Knobe
VI-A Company Thesis Supervisor

Certified by. .....
Daniel Jackson
M.I.T. Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students



Avalanche: A Framework for Parallel and Distributed Computation

by

Uriel Schafer

Submitted to the Department of Electrical Engineering and Computer Science
on Feb 1, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Existing frameworks for parallel and distributed programming either provide poor support for runtime flexibility or are overly restricted in their range of target applications. Here we describe the interface and implementation of the Avalanche framework, which attempts to solve both of these problems. In addition, we present the runtime performance of a canonical parallel algorithm written using Avalanche.

VI-A Company Thesis Supervisor: Kath Knobe

M.I.T. Thesis Supervisor: Daniel Jackson

Acknowledgments

I would like to thank:

My supervisor at Compaq, Kath Knobe, whose guidance in matters both thesis-related and personal was always available and always reliable.

My supervisor at MIT, Daniel Jackson, whose shining example of how a professor should care about his students inspires me every time we interact.

My colleagues at Compaq, George France and David Panariti who showed me the way through the technical quagmires of cluster computing.

My friends, Eric, Joe and Josh for keeping me sane and unproductive throughout my MIT career.

My siblings, Pella, Feivel, Reena and Shayna for always being there for me.

And, of course, my Abba and Eema. Without their constant love and support, nothing I have ever done would have been possible.

Contents

Chapter 1

Introduction

This chapter discusses the motivation for Avalanche.

Chapter two outlines the application programming interface (API) of Avalanche, and illustrates its use with some simple examples.

Chapter three describes the design of the runtime implementation of the Avalanche API.

Chapter four details the performance of a canonical parallel application written and run using Avalanche.

Chapter five concludes with some thoughts for future work.

1.1 Motivations for Avalanche

The motivation for Avalanche comes by way of comparison with existing parallel programming frameworks. These frameworks fall under two main categories:

- Message Passing Frameworks - such as PVM [1] and MPI [2]

- Data Parallel Frameworks - such as HPF [3]

1.2 Message Passing Frameworks

In message passing frameworks, all inter-node communication is explicitly defined by the application, as in the following example using PVM:

```
//Task 1
float a[1000][1000];            //Declare the array to be sent
pvm_initsend(<encoding>);       //Initialize the message buffer
pvm_pkfloat(a, 1000*1000, 1);   //Pack the array into the message buffer
pvm_send(2, <data-id>)];        //Send the message to Task 2

//Task 2
float a[1000][1000];            //Declare the array to be received
pvm_recv(1, <data-id>);         //Receive the message from Task 1
pvm_upkfloat(a, 1000*1000, 1)   //Unpack the message buffer into the array
```

From one perspective, the explicit nature of this framework is advantageous, as it allows applications to be more easily tuned for better performance on specific platforms, However, it also means that the application's communication patterns have to be defined statically, at compile time. As a result, they can't adapt to different cluster configurations, or to different load conditions at runtime.

This approach also has the negative side-effect of closely tying data-producing sections of the code with the data-consuming sections. Each must be intimately aware of the other's dataflow patterns to allow message coordination.

1.3 Data Parallel Frameworks

In data parallel frameworks, the programmer's task is much simpler. The application simply defines a data distribution pattern and all of the inter-node communication and coordination occurs "under the hood" as in the following HPF example:

```
REAL A(1000, 1000)                              !  Declare the shared array
PROCESSORS procs(4,4)                           !  Specify the processor grid
DISTRIBUTE(BLOCK, BLOCK) ONTO procs ::  A       !  Choose a data distribution
```

The compiler uses the directives provided by the application to distribute the data between the cluster's nodes and to coordinate its computation. While this scheme

greatly eases the burden on the application programmer, it is unsuited for applications with irregular data access patterns.

As well, while these directives work well at taking advantage of data-parallelism[1], they cannot address pipeline parallelism[2] or task parallelism[3].

1.4 The Avalanche Framework

The Avalanche model, by contrast, allows for task, data, and pipeline parallelism. In this model, data is implicitly distributed among the cluster nodes through the use of shared *channels*. All data stored and retrieved from these channels must be tagged with a unique identifier, but the data's producers and consumers require no knowledge of each other's internal workings.

Computation is also distributed in the Avalanche model. This is done by reducing the application to a series of iterated function calls, or *tasks*. These tasks are then characterized by their dependence on channel data, and assigned a distribution pattern suitable for a variety of cluster configurations (e.g. round-robin, node-specific, grid-based, etc...).

In the following Avalanche example, an iterated sorting task retrieves array data from a shared channel:

[1]Data Parallelism: Identical operations are simultaneously performed on different regions of the data.

[2]Pipeline Parallelism: Regions of data are pumped through a staged operation pipeline, with each stage operating simultaneously on a different data region.

[3]Task Parallelism: Unrelated operations are performed simultanously on possibly unrelated data.

```
void Sort(const StepId2& aStepId)
{
   //Extract the iteration indices from the StepId
   int i = aStepId[0];
   int j = aStepId[1];

   //Construct a DataId, and use it to retrieve
   //data from the DataStore channel
   DataId2 id(i, j);
   ConstDataItem<float[1000][1000]> unsorted = DataStore->Get(id);
   ...
}
```

The full Avalanche syntax will be explained in further detail later, but some key elements are visible in this short example. They are:

- Tasks as execution units - shown here in the iterated Sort function (the StepId argument identifies which iteration is to be executed).
- Channels used as storage devices - shown here in the DataStore channel (data is retrieved from the channel using a unique DataId identifier).

This combination of iterated execution and identifier-based storage allows for a natural description of the application's dataflow characteristics in a way conducive to distributed parallelization.

Chapter 2

The Avalanche API

Avalanche provides an application programming interface (API), which can be used directly by programmers, or as a target for code-generation utilities. The API is written in C++, and its discussion below assumes a solid command of that language.

2.1 Overview

The Avalanche API is divided into two sections:

- Application Definition: This section of the API defines the application's high-level structure (i.e. its storage and execution components). Objects such as Channels and Tasks are created using these methods.

- Runtime Information: This section of the API defines the runtime characteristics of the application (e.g. its dataflow patterns, execution dependencies, etc..). Objects such as StepHandlers and TaskInfos are used by these methods to define the application's characteristics.

These elements are explained in further detail below, but a summary of the API can be seen in figure 2-1.

Figure 2-1: The Avalanche API

2.2 Application Definition

Avalanche applications are defined in terms of their *Tasks* and *Channels*, which together make up a *Computation* object.

2.2.1 Storage

In Avalanche, storage is provided in units of *DataItems*. These DataItems are uniquely identified by *DataIds*, and managed using *Channels*.

DataItems

A DataItem is a handle to an array used by the application for storage or communication. The application specifies the type, dimensionality, and size of the array by including them as arguments to the DataItem template.

```
template <typename DataType>
struct DataItem {
...
```

Data managed by the DataItem can be accessed using the [ ] operator, following standard C++ notation, as in the following example:

```
//Define a type for the DataItems to be used
typedef DataItem<int[1000][1000]> Matrix;

//Allocate a new DataItem using a Channel
Matrix a = ...

//Modify elements of the DataItem like any other array
a[1][2] = a[3][4];
```

When DataItems are retrieved from a Channel object (see below) they are in the form of ConstDataItems. This means that their contained data can be observed using the [ ] operator, but can't be modified.

```
//Define a type for the ConstDataItems to be used
typedef ConstDataItem<int[1000][1000]> ConstMatrix;

//Retrieve a ConstDataItem from its Channel
ConstMatrix c = ...

//Access an element of the ConstDataItem:  OK
...  = c[3][4];

//Assign to an element of the ConstDataItem:  ILLEGAL
c[1][2] = ...
```

Note that in keeping with standard C++ semantics, data access is *not* checked for array bound violations.

Note too that since DataItems (and ConstDataItems) are only handles to their managed arrays, copying operations have to specify whether they are deep copies or shallow copies, as in the following example:

```
//Allocate a new DataItem using a Channel
Matrix a = ...

//Retrieve two ConstDataItems from their Channel
ConstMatrix c = ...
ConstMatrix d = ...

//Copy the <c> matrix into the <a> matrix (deep copy)
*a = *c;

//Copy the <d> handle into <c> (shallow copy)
c = d;
```

DataIds

DataIds are used to identify specific DataItems. These are n-tuples whose class name includes their length. Their elements can be accessed using the [ ] operator.

```
//Create two DataIds
DataId1 tinyDataId(1);
DataId7 longDataId(2, 3, 4, 5, 6, 7, 8);

//Mix and match their elements
tinyDataId[0] = longDataId[6];
```

Channels

Channels are used to store and retrieve DataItems using DataIds. The application specifies the types of the DataItems and the DataIds by supplying them as template arguments to the Channel template.

```
template <typename DataIdType, typename DataItemType>
class Channel
{
  public:
    //Item Handling
    DataItemType       NewItem(const DataIdType& aDataId);
    void               Put(const DataItemType& aItem);
    ConstDataItemType  Get(const DataIdType& aDataId);
};
```

DataItems are allocated using NewItem(), modified as necessary, and then Put() (committed) onto the Channel. Once committed, a DataItem can be retrieved in ConstDataItem form by using Get(). Modifying the original DataItem after it has

been committed has undefined behavior. The following example illustrates these uses:

```
//Define a type for the Channel to be used
typedef Channel<DataId2, Matrix> MatrixChannel;

//Create the Channel
MatrixChannel* DataStore = ...

//Allocate a DataItem with DataId <0, 0>
Matrix initial = DataStore->NewItem(DataId2(0, 0));

//Initialize the DataItem
for (int i = 0; i < 1000; ++i)
{
   for (int j = 0; j < 1000; ++j)
   {
      initial[i][j] = i*j;
   }
}

//Put the DataItem onto the Channel
DataStore->Put(initial);

//Modify the DataItem after it is Put() :  UNDEFINED
initial[0][0] = 1;
...

//Get the same DataItem back from the Channel (in const form)
ConstMatrix constInitial = MatrixChannel->Get(DataId2(0, 0));

//Access an element of the DataItem
int max = constInitial[999][999];
```

Avalanche makes the following assumptions about the application's use of these functions:

- Each DataId is assigned to at most one DataItem per Channel
- Any allocated DataItem is committed to its Channel exactly once.
- DataItems are retrieved from their Channel at most once per Task step (explained below).

These conditions help ensure that there are no DataId collisions, that DataItems are not needlessly duplicated, and that Avalanche's garbage collection functions prop-

erly. While some attempt to detect violation of these assumptions is made at runtime, not all violations will be detected. Those which remain undetected will lead to undefined behavior.

2.2.2 Execution

Execution in Avalanche is split into *Tasks*, which are executed in steps identified by *StepIds*.

StepIds

StepIds are used to identify individual steps of a Task (see below). Since a Task replaces what would be a loop nest in serial code, StepIds are needed to identify which step of that loop nest is currently executing. Like DataIds, StepIds are n-tuples whose class name indicates their length. The following example shows the evolution of a StepId:

```
//Original serial code
for (int i = 0; i < 1000; ++i)
{
   for (int j = 0; j < 1000; ++j)
   {
      //Do some work
      ...
   }
}

//Code converted into Task/StepId form
void Worker(const StepId2& aStepId)
{
   int i = aStepId[0];
   int j = aStepId[1];
   //Do the same work
   ...
}
```

Tasks

Task objects wrap around iterated functions like the Worker example shown above. The application specifies the length of the StepId used by the function (its iteration

vector) by supplying the StepId type as a parameter to the Task template:

```
template <typename StepIdType>
class Task
{
   public:
      typedef void (*WorkFunc)(const StepIdType&);
};
```

For the Worker example, the corresponding Task type would be:

```
typedef Task<StepId2> WorkerTask;
```

2.2.3 Computation

The application's Tasks and Channels, explained above, are created using a Computation object. This object is passed in uninitialized to an application-supplied AvalancheInit() function. There it is used to construct the application's components, using the templated NewChannel() and NewTask() functions, which are defined as follows:

```
class Computation
{
   public:
      template <typename ChannelType>
      ChannelType* NewChannel();

      template <typename TaskType>
      TaskType* NewTask(TaskType::WorkFunc& aFunc);
      ...
};
```

Using these functions, the application constructs its Tasks and Channels as in the following example:

```
//Channel which holds the data
static MatrixChannel* DataStore;

//Worker functions to be wrapped in Tasks
void Init(const StepId2& aStepId);
void Sort(const StepId2& aStepId);
void Output(const StepId2& aStepId);

//Computation construction
void AvalancheInit(Computation& aComp)
{
   DataStore = aComp.NewChannel<MatrixChannel>();
   WorkerTask* initTask = aComp.NewTask<WorkerTask>(Init);
   WorkerTask* sortTask = aComp.NewTask<WorkerTask>(Sort);
   WorkerTask* outputTask = aComp.NewTask<WorkerTask>(Output);
   ...
}
```

The application also uses the Computation class to indicate that it has completed by invoking the static Computation::Halt() method as follows:

```
void FinalTask(...)
{
   //...Finish off the last bits of work

   Computation::Halt();
}
```

2.3 Runtime Information

Applications provide two types of information to the Avalanche runtime, their dataflow patterns and their execution patterns.

2.3.1 Dataflow Patterns

The application's dataflow patterns are communicated to the Avalanche runtime by creating *Edges* between Channels and the Tasks which use their data. The details of the Edges' dataflows are indicated by enabling functions which pass the information through *StepHandlers*.

StepHandlers

StepHandlers pass dataflow information about Tasks to the Avalanche runtime. The exact meaning of the information is context-dependent, but it is passed on an individual step basis by using the StepHandler's AddStep() method. To accomplish this, the StepHandler's types are parameterized on the Task whose steps they handle.

```
template <typename TaskType>
class StepHandler
{
   public:
      //Step Notification
      virtual void AddStep(const StepIdType& aStep) = 0;
};
```

The most common information passed through StepHandlers is: "Which Task steps are enabled by this DataItem?" (i.e. "Which Task steps need this DataItem to run?").

For example, consider the following sorting task:

```
//Each step of this sorting task needs a DataItem from DataStore
void Sort(const StepId2& aStepId)
{
   int i = aStepId[0];
   int j = aStepId[1];
   ConstMatrix unsorted = DataStore->Get(DataId2(i, j));
   ...
}
```

Here, each DataItem inserted into DataStore with the DataId <i, j>, will be needed by the step of Sort with StepId {i, j}. The application would pass this dataflow information to the Avalanche runtime via a StepHandler-using enabling function similar to this:

```
//Each DataItem produced on DataStore enables
//the corresponding step of Sort
void DataStoreToSort(const DataId2& aDataId,
                     const StepHandler<WorkerTask>& aHandler)
{
   int i = aDataId[0];
   int j = aDataId[1];
   aHandler.AddStep(StepId2(i, j));
}
```

Edges

When, as in the above example, dataflow from a Channel to a Task is needed for that Task to execute, an Edge between the two is created in the Computation object. That Edge is created in the application's AvalancheInit() function using the NewEdge() method of the Computation object. This method is defined as follows:

```
class Computation
{
   ...
   public:
      template <typename ChannelType, typename TaskType>
      void NewEdge(ChannelType& aChannel,
                   TaskType&    aTask,
                   EnableFunc&  aFunc);
};
```

Using this function, the application connects its Channels to the Tasks which use their data as in the following example:

```
void AvalancheInit(Computation& aComp)
{
   ...
   aComp.NewEdge(*DataStore, *sortTask, DataStoreToSort);
}
```

2.3.2 Execution Patterns

The execution patterns of the application's Task steps are defined by their locations and running requirements. These are communicated to the Avalanche runtime by attaching a *TaskInfo* object to each Task.

Step Location

The execution location of each Task step is specified using locator functions, which are defined as:

```
typedef NodeId (*LocateFunc) (  const StepId& aStepId
                                const Cluster& aCluster);
```

These functions return the NodeId which specifies the execution location for a given StepId and Cluster configuration, as in the following examples:

```
//Each step of Output is executed on the same node in the Cluster
NodeId OutputLocator(const StepId2& aStepId, const Cluster& aCluster)
{
   return 0;
}

//The steps of Sort are executed in round-robin order
//on the Cluster's nodes
NodeId ModLocator(const StepId2& aStepId, const Cluster& aCluster)
{
   return aStepId.Hash() % aCluster.SizeInNodes();
}

//The steps of Init are split into blocks along the
//first index, and divided between the Cluster's nodes
NodeId BlockLocator(const StepId2& aStepId, const Cluster& aCluster)
{
   return aStepId[0] * aCluster.SizeInNodes() / N;
}
```

For convenience, these three cases are generalized and provided as the templated functions ConstLocator, ModLocator, and BlockLocator. Using the templates, the functions above would appear as:

```
//Each step of Output is executed on the same node in the Cluster
ConstLocator<WorkerTask, 0>;

//The steps of Sort are executed in round-robin order
//on the Cluster's nodes
ModLocator<WorkerTask>;

//The steps of Init are split into blocks along the
//first index, and divided between the Cluster's nodes
BlockLocator<WorkerTask, 0, N>;
```

Step Running

The running requirements of a Task's steps are specified in two different ways depending on the type of Task.

Waiting Tasks

Most Tasks require some number of DataItems to be available before they can run. These Tasks use a waiting function to specify how many DataItems each step waits for. Waiting functions are defined as:

```
typedef unsigned int (*WaitFunc) (const StepId& aStepId);
```

These functions simply return the number of DataItems which a particular step waits for, as in the following example:

```
//Each step of Sort requires one DataItem to run
unsigned int SortWaiter(const StepId2& aStepId)
{
   return 1;
}
```

For convenience, this case is generalized and provided as the templated function ConstWaiter. Using the template, the function above would appear as:

```
//Each step of Sort requires one DataItem to run
ConstWaiter<WorkerTask, 1>;
```

Ranging Tasks

Some Tasks, however, have steps which don't need any DataItems to run. These "ranged" Tasks, typically initializers, use a ranging function to delimit their range of valid steps. Ranging functions are defined as:

```
typedef void (*RangeFunc) (const StepHandler<TaskType>& aHandler);
```

These functions take in a StepHandler, and add to it all of the steps which are within the valid range, as in the following example:

```
//Init runs from {0, 0} to {N-1, N-1}
void InitRanger(const StepHandler<WorkerTask>& aHandler)
{
   for (int i = 0; i < N; ++i)
   {
      for (int j = 0; j < N; ++j)
      {
         aHandler.addStep(StepId2(i, j));
      }
   }
}
```

TaskInfos

Once the locating and run-enabling functions of each Task have been defined, they are combined to form a TaskInfo. TaskInfos are parameterized on the type of the Task they have information about.

```
template <typename TaskType>
class TaskInfo {
   ...
```

Once created, the TaskInfos are inserted into their respective Tasks, as shown in the following example:

```
void AvalancheInit(Computation& aComp)
{
  ...
  initTask->SetInfo(TaskInfo<WorkerTask>(    InitRanger,
                                             BlockLocator<WorkerTask, 0, N>));
  sortTask->SetInfo(TaskInfo<WorkerTask>(    ConstWaiter<WorkerTask, 1>,
                                             ModLocator<WorkerTask>));
  outputTask->SetInfo(TaskInfo<WorkerTask>(  ConstWaiter<WorkerTask, N>,
                                             ConstLocator<WorkerTask, 0>));
}
```

Having specified all of the above information, the application is complete and ready to be executed by the Avalanche runtime, detailed in the next chapter. For reference purposese, an example of a complete Avalanche application can be found in Appendix A.

Chapter 3

Avalanche Runtime

As we saw in the previous chapter, the application uses the Avalanche API to:

- define its execution in terms of Tasks,
- define its storage in terms of Channels and DataItems,
- specifiy the dataflow between Channels and Tasks,
- specify the distribution pattern for its Tasks,
- and indicate when it has finished by calling Computation::Halt().

In return, the Avalanche runtime:

- spawns the application on all of the available cluster nodes,
- distributes execution between those nodes according the the specified pattern,
- enables execution as data becomes available,
- distributes the data through the cluster,
- and garbage collects the data when it is no longer needed.

In this chapter, we will discuss the design of the Avalanche runtime, focussing on its object model and execution strategy, including some of the more consequential decisions made during the design process.

3.1 Object Model

The Avalanche runtime's object model follows naturally from its API, as can be seen from the slightly simplified extract in Figure 3-1. While the overall structure is fairly self-explanatory, the individual classes have various interesting features which are explained below.

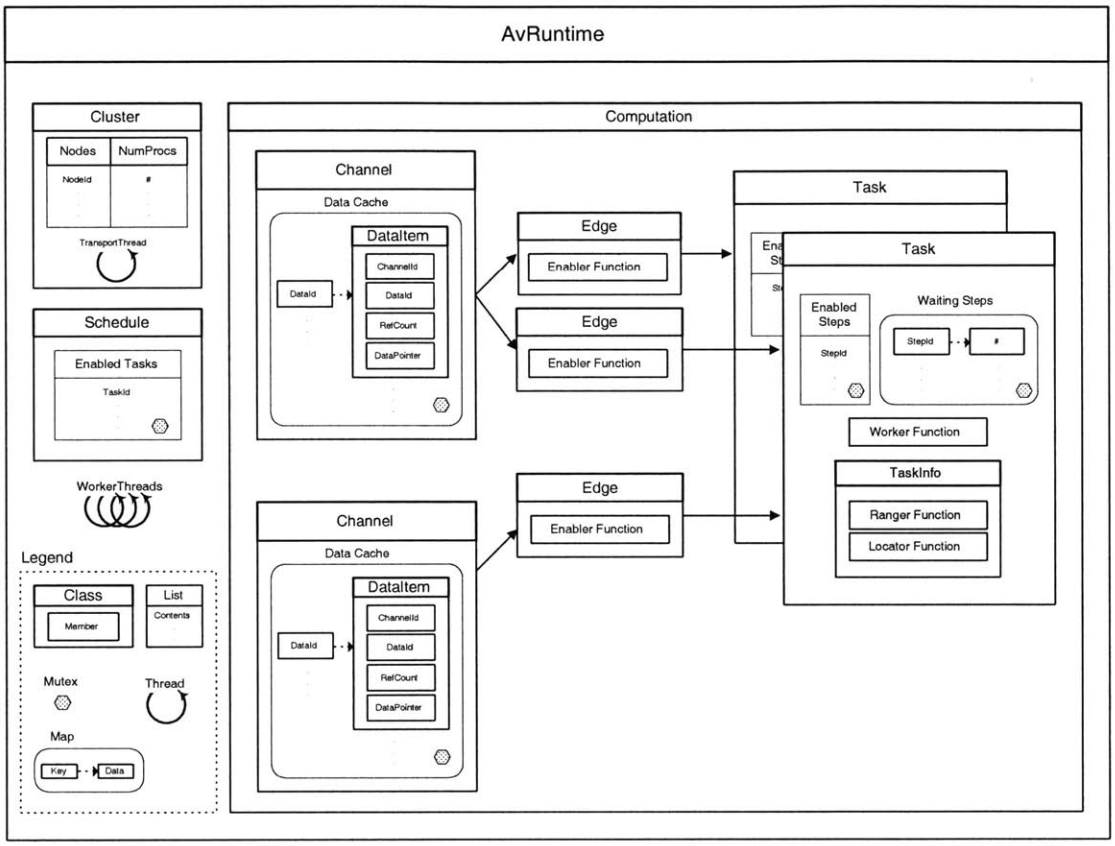


Figure 3-1: Avalanche Runtime Object Model

3.1.1 AvRuntime

The AvRuntime object encapsulates the Avalanche runtime and acts as an intermediary between its components - the cluster model, the computation model, the task schedule, and the worker threads.

Despite the fact that the Avalanche runtime is multithreaded, only one AvRuntime object is used per node, so its mutable components must be protected through the

use of mutices. While this imposes a certain degree of synchronization overhead, it allows the various worker threads to share channel caches and to load-balance tasks.

3.1.2 WorkerThreads

The actual work of the application is performed by pthreads-based[4] WorkerThreads, where a different WorkerThread runs on each processor. These WorkerThreads share the same task schedule, which allows them to load-balance between tasks being executed on the same node.

3.1.3 Schedule

The task schedule is implemented as a LIFO queue of task identifiers. The identified tasks can (and usually do) appear more than once in the schedule, indicating that they have more than one enabled step waiting for execution.

The worker threads use the schedule by popping task identifiers off of it, and executing those tasks' next enabled step. This two-stage step execution process imposes additional synchronization overhead, but has the benefit of allowing the schedule to not store the task's step-identifiers, and instead be independent of their types.

3.1.4 Cluster

The Cluster object's main function is to maintain an up-to-date model of the current cluster configuration. While further information might be needed at some later date, this configuration is currently just a list of node identifiers and a count of the processors on each node.

As well, the Cluster object contains the pthreads-based TransportThread, which is responsible for incoming communication via the underlying PVM[1] message-passing library. To handle this, the thread polls regularly for message arrivals, and inserts any received data-items into their parent channels.

3.1.5 Channel

The globally shared Channels implied by the Avalanche API are implemented by using local Channel objects as caches, and only storing in them the data-items which will be needed at the local node. By limiting the number of data-items stored at each node, we allow the cache storage to be implemented using in-memory hash-maps, which map the data-id to the actual data-item. This strategy works due to the write-once nature of the API, which guarantees that data-items only need to be distributed once - when they are committed - to provide all nodes with a consistent view of the data-space.

Channels also implement Avalanche's garbage-collection policy by computing and storing reference counts with each created data-item. When a Get() occurs, the reference count is decremented[1], and the data-item's storage is freed when the count reaches zero.

3.1.6 DataItem

DataItems contain a pointer to their managed data and a reference count, as would be expected. As well, they contain a copy of their own identifier and that of their channel. This redundancy allows for a simpler Put() interface, and simplies the DataItem's conversion to a transport message.

Task

The Task object contains two collections of task-steps in addition to its worker function and task-information. The first of these collections is a LIFO queue of ready-to-run steps, from which the WorkerThreads extract steps for execution. The second is a hash-map which maps a step's identifier to the count of data-items it is still waiting for. These two collections are used in implementing Avalanche's data-driven step-enabling strategy.

[1]To avoid premature garbage collection, the runtime doesn't actually decrement the reference count immediately. Instead, the read-only data-item returned by Get() is implemented as a temporary, whose destruction (at the end of the basic block where the Get() occurs) triggers the decrement.

Rather than tracking the enabled status of all valid steps of a given Task, the Avalanche runtime monitors only those steps which are at least partially enabled. When an additional data-item needed by some steps is inserted into its channel, the runtime determines which steps are waiting for it by using the channel's enabling functions. The runtime then uses the Task's waiting function to determine the steps' total data requirements, creates or adjusts the steps' entries in the waiting-step collection to reflect the newest data, and moves the steps to the enabled queue if appropriate.

This data-driven approach has two benefits:

- By selecting the channel insertion as the enabling trigger, this approach allows both locally created data and remotely created data to trigger the enabling process in near-identical fashion.

- By enabling the steps just as new data arrives, and feeding the steps out in LIFO order, this strategy achieves a fair amount of data locality without the need for a sophisticated scheduler.

3.2 Execution Overview

The Avalanche runtime manages the application's execution, utilizing the components explained above, by following these steps:

1. The user starts the application on an arbitrary node of the cluster.

2. The linked-in initialization sequence creates a Cluster object, which spawns child applications on the other nodes in the cluster.

3. The parent Cluster object waits for all of the children to finish spawning, then broadcasts a "begin" message.

4. Each node's initialization sequence creates its own AvRuntime object, passing in the Cluster model.

5. The AvRuntime object calls AvalancheInit(), starts the TransportThread, starts the WorkerThreads, and waits for the threads to complete.

6. The runtime executes the application's Task steps, creating and distributing DataItems throughout the cluster where they trigger the execution of other Task steps.

7. Computation::Halt() is called by the application, causing "end" messages to be broadcast to all of the cluster's nodes. This causes the WorkerThreads and TransportThreads to exit, terminating the application.

This execution life-cycle minimizes communication between the cluster's nodes, and requires cluster-wide barriers only at startup and shutdown.

3.3 Design Decisions

In the process of finalizing Avalanche's design, various decisions were made based on a variety of rationales. Here we elaborate on several of these decisions, focussing on those which had wide-reaching effects on the runtime's final implementation.

3.3.1 Templates

One of the more controversial decisions in the design process was our heavy use of C++ templates. These templates significantly decrease the code's readability, and severely limit our ability to link with applications written in other languages. However, the alternative - using *void** and leaving type-safety entirely in the hand of the user - was so unappealing that we chose the templated route instead. While the type-checking provided as a result *has* caught several errors, it remains to be seen if this decision was ultimately correct.

3.3.2 Step List Transmission

When a data-item is created in Avalanche, the runtime must generate a list of all the steps enabled by that data-item before filtering the list so that only the local ones remain. In the original design for the Avalanche runtime, the remote steps on the list were not discarded, but were instead sent along with their enabling data-item to the nodes where they would be executed.

This allowed us the benefit of avoiding the inefficient recomputing of the lists at the other end. We soon realized, however, that the cost of transmitting the lists far outweighed the efficiency benefits of having a single point of computation, and changed the implementation to discard them instead.

3.3.3 StepHandlers

While the Avalanche API is fairly easy to grasp once its model of execution is understood, StepHandlers remain difficult to comprehend even after that point. Their inclusion in the API resulted from the realization that in order to return multiple task-steps from an enabling function, the function would have to iterate through those return values. If it inserted them into a collection during that iteration, the caller would be forced to iterate through the collection again to extract them.

Instead, StepHandlers are passed by the caller into the enabling function, and the callee uses AddStep() to execute a context-specific callback which appropriately handles the step. While this results in a cleaner and more efficient implementation, it is not clear that the efficiency gains justify the accompanying loss of clarity in the API.

Chapter 4

Performance

To test the performance, scalability, and usability of the Avalanche framework, we implemented four parallel algorithms using the API, and ran them on two four-way Alpha SMP's.

4.1 Algorithms

The implemented algorithms were chosen in an attempt to include representatives of different algorithm categories (divide-and-conquer, iterative relaxation, etc.). In addition, algorithms with irregular data-access patterns were selected, in an attempt to highlight Avalanche's capabilities.

Based on these criteria, we chose:

- Fast Fourier Transform
- LU Decomposition
- Merge Sort
- Jacobian Relaxation

In implementing these algorithms, we discovered that Avalanche is rather sensitive to the amount of work done in each task step. Too much work, and Avalanche isn't able to take full advantage of the exposed parallelism. To little work, and the

synchronization overhead of running multiple threads outweighs the parallelism gains. As a result of this sensitivity, coupled with time constraints, we were only able to tune the relaxation application to the degree needed to achieve the proper SMP scaling.

4.2 Setup

We ran the relaxation algorithm over a 15000 by 300 grid of integers for 30 ticks. We were able to measure Avalanche's scaling across two Alpha SMPs by configuring the PVM daemon to vary the number of processors which it made available to Avalanche. The first four processors exposed were on the host node, while the second four were on a second node in the cluster. The application was run five times at each of the eight possible configurations, and the average runtime of the five was chosen as the outcome. The results are shown in Figure 4-1, along with a plot of the optimal linear speed-up, shown for reference.

4.3 Interpretation

Two points are of significance when viewing these results:

4.3.1 SMP Scaling

Avalanche scaled admirably well when additional processors were made available to it on a single SMP. While the optimal linear scaling was not achieved, the performance gains were roughly comparable.

4.3.2 Multinode Scaling

Here, unfortunately, Avalanche did not fare as well. At the point where the PVM daemon made available the fifth processor (located on the second SMP) performance actually decreased. A slight gain was seen from the sixth processor, but performance quickly worsened when the last two remote processors were made available.

Upon examining the data-access patterns of the application, it became clear that the underlying PVM transport was the weak link. It was apparently so inefficient that it had become more expensive to ship the data then to perform all the computation locally. By dividing the data (and computation) evenly between the exposed processors, Avalanche was allocating increasing percentages to separate nodes, driving up the communication costs which outweighed the computation benefits.

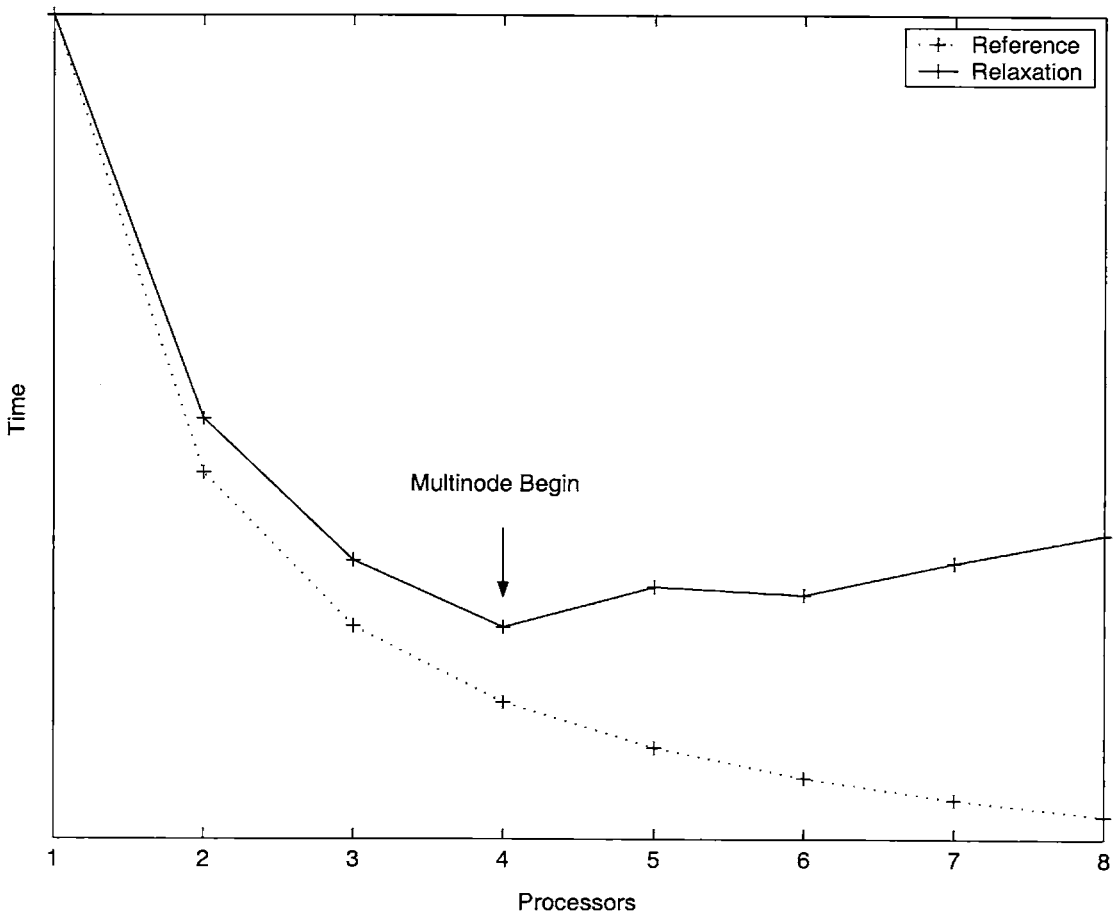


Figure 4-1: Relaxation Results

Chapter 5

Future Work

While Avalanche has managed to reach its initial goals of a stable API and SMP scalibility, there is still a great deal of work to be done in order to broaden its range of usability.

5.1 Transport

First and foremost, the inefficiency of the current transport must be addressed. While the near-linear scaling of Avalanche on a single SMP is encouraging, its inability to fully take advantage of other nodes in the cluster leaves Avalanche with a sizable gap in functionality. Reimplementing Avalanche to use another underlying transport (e.g. MPI), would be a viable alternative, should PVM prove to not be sufficiently tunable.

5.2 Dynamic Reconfiguration

An additional consideration during the design of the Avalanche runtime was allowance for a dynamic reconfiguration of the cluster while an application is running. This would allow computational resources to be added to the cluster when idle, but removed when needed for another task. While the algorithm for doing this was outlined, and hooks provided for its implementation in the runtime, time constraints prevented

its completion.

5.3 Automatic Generation

Finally, the Avalanche project was ultimately intended not as a target for programmers, but as a target for compilers. While some parts of the conversion algorithm are still to be formalized, a preliminary explanation for automatic generation of an Avalanche application from serial code has been completed. It remains to be seen whether the remaining pieces will prove tractable, but the results are encouraging thus far, and promise to extend the range of Avalanche's applicability immensely.

Appendix A

Sample Avalanche Application

The following is a sample application written using the Avalanche API. This "embarrassingly parallel" application initializes, sorts, and outputs 1,000 by 1,000 matrices of a 1,000,000 by 1,000,000 integer array. The initialization and sorting are distributed in round-robin fashion around the cluster, and the output is done entirely on node 0.

```
#include "Avalanche.h"

//Application Constants
const static int N = 1000;

//Type definitions
typedef DataItem<int[1000][1000]> Matrix;
typedef ConstDataItem<int[1000][1000]> ConstMatrix;
typedef Channel<DataId2, Matrix> MatrixChannel;
typedef Task<StepId2> WorkerTask;

//Channel declarations
static MatrixChannel* DataStore;
static MatrixChannel* OutputStore;

//Worker functions to be wrapped in Tasks
void Init(const StepId2& aStepId)
{
   Matrix initial = DataStore->NewItem(DataId2(aStepId[0], aStepId[1]));

   //...Initialize data in initial

   DataStore->Put(initial);
}
```

```
void Sort(const StepId2& aStepId)
{
   ConstMatrix unsorted = DataStore->Get(DataId2(aStepId[0], aStepId[1]));
   Matrix sorted = OutputStore->NewItem(DataId2(aStepId[0], aStepId[1]));

   //...Sort unsorted and insert results into sorted

   OutputStore->Put(sorted);
}

void Output(const StepId2& aStepId)
{
   ConstMatrix sorted = OutputStore->Get(DataId2(aStepId[0], aStepId[1]));

   //...Output the sorted matrix

   Computation::Halt();
}

//Dataflow functions
void DataStoreToSort(const DataId2& aDataId,
                     const StepHandler<WorkerTask>& aHandler)
{
   aHandler.AddStep(StepId2(aDataId[0], aDataId[1]));
}

void OutputStoreToOutput(const DataId2& aDataId,
                         const StepHandler<WorkerTask>& aHandler)
{
   aHandler.AddStep(StepId2(aDataId[0], aDataId[1]));
}

//Ranging function
void InitRanger(const StepHandler<WorkerTask>& aHandler)
{
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
      {
         aHandler.AddStep(StepId2(i, j));
      }
   }
}
```

```
//Computation construction
void AvalancheInit(Computation& aComp)
{
   //Construct the Channels
   DataStore = aComp.NewChannel<MatrixChannel>();
   OutputStore = aComp.NewChannel<MatrixChannel>();

   //Construct the Tasks
   WorkerTask* initTask = aComp.NewTask<WorkerTask>(Init);
   WorkerTask* sortTask = aComp.NewTask<WorkerTask>(Sort);
   WorkerTask* outputTask = aComp.NewTask<WorkerTask>(Output);

   //Insert the TaskInfos
   initTask->SetInfo(TaskInfo<WorkerTask>(InitRanger,
                                          ModLocator<WorkerTask>));
   sortTask->SetInfo(TaskInfo<WorkerTask>(ConstWaiter<WorkerTask, 1>,
                                          ModLocator<WorkerTask>));
   outputTask->SetInfo(TaskInfo<WorkerTask>(ConstWaiter<WorkerTask, 1>,
                                            ConstLocator<WorkerTask, 0>));

   //Construct the Edges
   aComp.NewEdge(*DataStore, *sortTask, DataStoreToSort);
   aComp.NewEdge(*OutputStore, *outputTask, OutputStoreToOutput);
}
```

Bibliography

[1] Sunderam, V. *PVM: A Framework for Parallel Distributed Computing.* Concurrency: Practice and Experience, 2(4):315–339, December 1990.

[2] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference.* MIT Press, 1995.

[3] High Performance Fortran Forum. *High Performance Fortran Language Specification.* Scientific Programming, 2:1–170, 1993.

[4] "Pthreads: POSIX threads standard", *IEEE Standard* 1003.1c-1995.